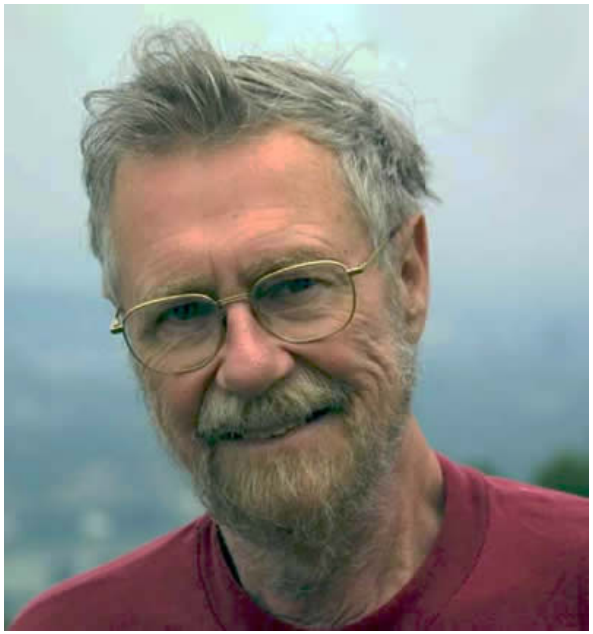
Clases en Python: lo estás haciendo mal

Víctor Terrón — http://github.com/vterron

Programación orientada a objetos

Una cita apócrifa

"Object-oriented programming is an exceptionally bad idea which could only have originated in California" [Edsger W. Dijkstra]



Programación orientada a objetos

Lo que en realidad dijo

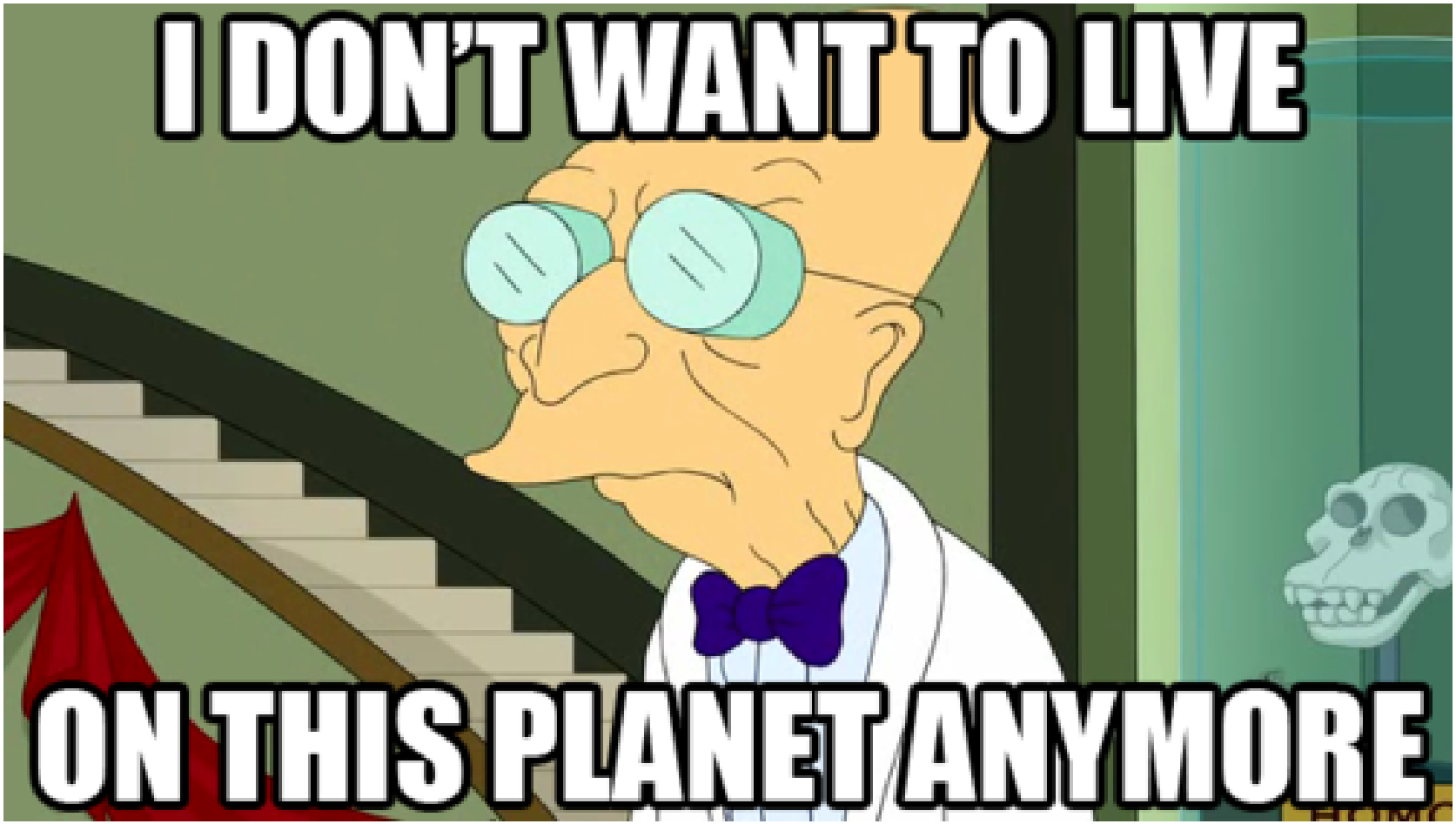
"For those who have wondered: I don't think object-oriented programming is a structuring paradigm that meets my standards of elegance." [Edsger W. Dijkstra, 1999] [Fuente]

Programando clases en Python

El objetivo de esta charla es ayudarnos a hacer idiomáticas y elegantes nuestras clases en Python.

En mayor o menor medida todo el mundo usa programación orientada a objetos, pero hay una serie de aspectos fundamentales que es indispensable conocer para evitar que nuestro código sea innecesariamente feo o complejo.

Nuestra reacción a veces



Programando clases en Python

La premisa es que somos mínimamente familiares con los conceptos básicos de programación orientada a objetos, y que hemos trabajado un poco con nuestras propias clases en Python.

Un muy brevísimo repaso

Uno

Llamamos clase a la representación abstracta de un concepto. Por ejemplo, 'perro', 'número entero' o 'servidor web'.

Dos

Las clases se componen de atributos y métodos.

Tres

Un objeto es cada una de las instancias de una clase.

Ejemplo: clase Perro

```
1  #! /usr/bin/env python
2  # encoding: UTF-8
3
4  class Perro(object):
5
6      def __init__(self, nombre, raza, edad):
7          self.nombre = nombre
8          self.raza = raza
9          self.edad = edad
10
11     def ladra(self):
12         print self.nombre, "dice '¡Wooof!'"
13
14 mascota = Perro("Lassie", "Collie", 18)
15 mascota.ladra()
```

¡No os limitéis a escuchar!

No suele ser divertido escuchar a nadie hablar durante casi una hora. Participad, intervenid, criticad, opinad. ¡Si digo algo que no tiene ningún sentido, corregidme!

Transparencias y código fuente en:

`http://github.com/vterron/PyConES-2014`

Erratas, correcciones, enlaces, ¡cualquier cosa!

Índice

Índice

00. Heredando de object (new-style)

Hemos de heredar de object para que nuestra clase sea *new-style*.

```
1  class Perro(object):
2
3      def __init__(self, nombre, raza, edad):
4          self.nombre = nombre
5          self.raza = raza
6          self.edad = edad
7
8  mascota = Perro("Lassie", "Collie", 18)
9  print "Clase:", mascota.__class__
10 print "Tipo:", type(mascota)
```

```
Clase: <class '__main__.Perro'>
Tipo: <class '__main__.Perro'>
```

00. Heredando de object: por qué

Algo de historia

Las clases *new-style* aparecieron en Python 2.2 (diciembre de 2001)

Hasta Python 2.1, el concepto de clase no tenía relación con el tipo: los objetos de todas las (*old-style*) clases tenían el mismo tipo: <type 'instance' >. Las clases *new-style* unifican clase y tipo — dicho de otra forma: una clase *new-style* no es sino un tipo definido por el usuario.

00. Heredando de object: ejemplo

```
1  class Perro: # no heredamos de 'object'
2
3      def __init__(self, nombre, raza, edad):
4          self.nombre = nombre
5          self.raza = raza
6          self.edad = edad
7
8  sargento = Perro("Stubby", "Bull Terrier", 9)
9  print "Clase:", sargento.__class__
10 print "Tipo:", type(sargento)
```

```
Clase: __main__.Perro
Tipo: <type 'instance'>
```

00. La vida antes de Python 2.2

Pero no es una cuestión sólo de qué devuelve type()

- No existía super()...
- ... ni los descriptores...
- ... ni __slots__
- El MRO (*Method Resolution Order*) era mucho más simple.
- Las clases podían ser lanzadas, sin heredar de Exception.

Old style and new style classes in Python

`https://stackoverflow.com/a/19950198/184363`

00. Lanzando una clase *new-style*

```
1 class Spam(object):
2     pass
3
4 raise Spam()
```

```
Traceback (most recent call last):
  File "./code/00/02-new-style-raise.py", line 4, in <module>
    raise Spam()
TypeError: exceptions must be old-style classes or derived from
BaseException, not Spam
```

00. Lanzando una clase *old-style*

```
1 class Foo:
2     pass
3
4 raise Foo()
```

```
Traceback (most recent call last):
  File "./code/00/03-old-style-raise.py", line 4, in <module>
    raise Foo()
__main__.Foo: <__main__.Foo instance at 0x402b948c>
```

00. Heredando de object (new-style)

- Por compatibilidad, las clases siguen siendo *old-style* por defecto.
- En Python 3 las clases *old-style* han desaparecido — heredar de object es opcional.
- El resumen es que tenemos que heredar siempre de object (o de otra clase *new-style*, claro).

Moraleja

Hay que heredar de object *siempre*

Índice

01. Python's super() considered super!

La función super() nos devuelve un objeto proxy que delega llamadas a una superclase.

En su uso más básico y común, nos permite llamar a un método definido en la clase base (es decir, de la que hemos heredado) sin tener que nombrarlas explícitamente, haciendo nuestro código más sencillo y mantenible.

01. Algo que *no* funciona

```
1  # encoding: UTF-8
2
3  class ListaLogger(list):
4      def append(self, x):
5          print "Intentando añadir", x
6          self.append(x)
7
8  numeros = ListaLogger()
9  numeros.append(7)
```

```
Intentando añadir 7
Intentando añadir 7
Intentando añadir 7
Intentando añadir 7
Intentando añadir 7
Intentando añadir 7
Intentando añadir 7
Intentando añadir 7
```

01. Hay que hacerlo así

```
1  # encoding: UTF-8
2
3  class ListaLogger(list):
4      def append(self, x):
5          print "Añadiendo", x, "a la lista (¡ahora sí!)"
6          super(ListaLogger, self).append(x)
7
8  numeros = ListaLogger()
9  numeros.append(7)
10 print numeros
```

```
Añadiendo 7 a la lista (¡ahora sí!)
[7]
```

01. Un segundo ejemplo

```
1  class Punto(object):
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6  class Circulo(Punto):
7      def __init__(self, x, y, radio):
8          super(Circulo, self).__init__(x, y)
9          self.radio = radio
10
11 c = Circulo(3, 4, 5.5)
12 print "x:", c.x
13 print "y:", c.y
14 print "r:", c.radio
```

```
x: 3
y: 4
r: 5.5
```

01. super() en Python 3000

En Python 3 ya no es necesario especificar la clase actual.

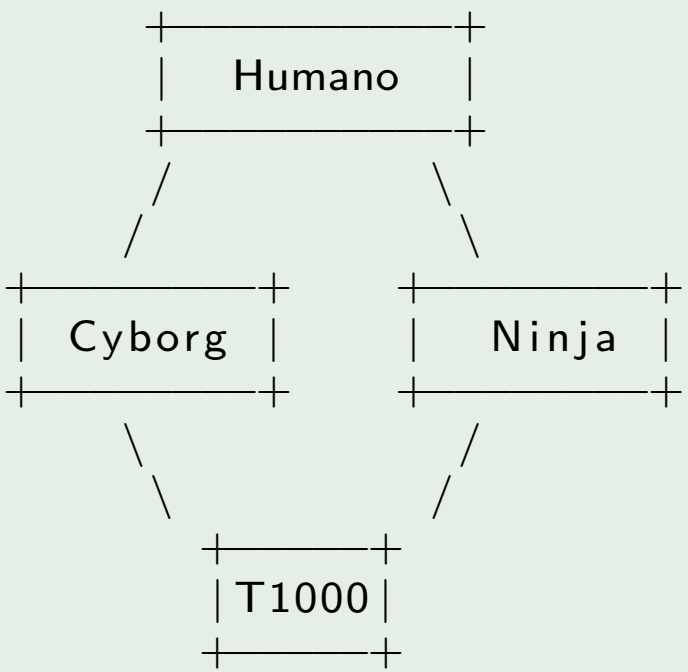
```
1  class ListaLogger(list):
2      def append(self, x):
3          print("Esto también añade", x)
4          super().append(x)
5
6  numeros = ListaLogger([4, 5])
7  numeros.append(-1)
8  print(numeros)
```

```
Esto también añade -1
[4, 5, -1]
```

01. Herencia múltiple

Donde el verdadero poder de super() se hace patente, sin embargo, es cuando trabajamos con herencia múltiple.

Diagrama de clases en texto plano cutre



01. T-1000

```
1  # encoding: UTF-8
2
3  class Humano(object):
4      def ataca(self):
5          print "Puñetazo"
6
7  class Cyborg(Humano):
8      def ataca(self):
9          print "Láser"
10
11 class Ninja(Humano):
12     def ataca(self):
13         print "Shuriken"
14
15 class T1000(Cyborg, Ninja):
16     def ataca(self, n):
17         for _ in xrange(n):
18             super(T1000, self).ataca()
19
20 print "MRO:", T1000.__mro__
21 robot = T1000()
22 robot.ataca(5)
```

01. T-1000

Usamos super() para llamar al método padre de T1000.

```
MRO: (<class '__main__.T1000'>, <class '__main__.Cyborg'>, <class
'__main__.Ninja'>, <class '__main__.Humano'>, <type 'object'>)
Láser
Láser
Láser
Láser
Láser
```

01. T-1000

Hemos visto que T1000 hereda antes de Cyborg que de Ninja, por lo que Cyborg delega la llamada en Cyborg.ataca(). ¿Y si Cyborg no define ataca()?

Podríamos pensar que entonces Cyborg buscará el método en su padre: Humano.

01. T-1000

```
1  # encoding: UTF-8
2
3  class Humano(object):
4      def ataca(self):
5          print "Puñetazo"
6
7  class Cyborg(Humano):
8      pass
9
10 class Ninja(Humano):
11     def ataca(self):
12         print "Shuriken"
13
14 class T1000(Cyborg, Ninja):
15     def ataca(self, n):
16         for _ in xrange(n):
17             super(T1000, self).ataca()
18
19 robot = T1000()
20 robot.ataca(5)
```

01. Clases base y hermanas

```
Shuriken
Shuriken
Shuriken
Shuriken
Shuriken
```

Pero no. Lo que super() hace es recorrer el MRO y delegar en la primera clase que encuentra por encima de T1000 que define el método que estamos buscando.

Es el hermano de Cyborg, Ninja, quien define ataca()

01. Superclases y herencia múltiple

El concepto de superclase no tiene sentido en herencia múltiple.

Esto es lo terrorífico de super(), y a su vez tan maravilloso: nos permite construir subclases que componen a otras ya existentes, determinando su comportamiento según el orden en el que se heredan.

01. super() — el orden importa

```
1  # encoding: UTF-8
2
3  class Humano(object):
4      def ataca(self):
5          print "Puñetazo"
6
7  class Cyborg(Humano):
8      def defiende(self):
9          print "Armadura"
10
11 class Ninja(Humano):
12     def ataca(self):
13         print "Shuriken"
14
15 class T1000(Cyborg, Ninja):
16     pass
17
18 robot = T1000()
19 robot.defiende()
20 robot.ataca()
```

01. super() — el orden importa

```
Armadura
Shuriken
```

Python's super() considered super!

https://rhettinger.wordpress.com/2011/05/26/

Things to Know About Python Super [1 of 3]

http://www.artima.com/weblogs/viewpost.jsp?thread=236275

Understanding Python super() and __init__() methods

https://stackoverflow.com/q/576169/184363

01. Python's super() considered super!

En opinión de Michele Simionato, super() es uno de los detalles de Python más complicados de entender ("*one of the most tricky and surprising Python constructs*")

Moraleja

Usamos super() para llamar a métodos definidos en alguna de las clases de las que heredamos.

Índice

02. El primer parámetro: self

Al definir el método de una clase, el primer parámetro ha de ser self: esta variable es una referencia al objeto de la clase que ha llamado al método.

Es equivalente al this de otros lenguajes como Java o C++.

01. Ejemplo: __init__() y distancia()

```
1  import math
2
3  class Punto(object):
4      def __init__(self, x, y):
5          self.x = x
6          self.y = y
7
8      def distancia(self, otro):
9          x_delta = self.x - otro.x
10         y_delta = self.y - otro.y
11         return math.sqrt(x_delta ** 2 + y_delta ** 2)
12
13 c1 = Punto(8, 1)
14 c2 = Punto(4, 7)
15 print c1.distancia(c2)
```

```
7.21110255093
```

02. El nombre 'self' es una convención

```
1  class Punto(object):
2      def __init__(this, x, y):
3          this.x = x
4          this.y = y
5
6  class Circulo(Punto):
7      def __init__(this, x, y, radio):
8          super(Circulo, this).__init__(x, y)
9          this.radio = radio
10
11 c = Circulo(2, 7, 5)
12 print "x:", c.x
13 print "y:", c.y
14 print "r:", c.radio
```

```
x: 2
y: 7
r: 5
```

02. ¿Por qué tenemos que definirlo?

El parámetro no tiene por qué llamarse 'self': puede ser cualquier otra cosa. Pero self es la convención, y culturalmente está mal visto usar algo diferente.

¿Pero por qué tenemos que añadirlo explícitamente a cada método que definamos? ¿Por qué no puede declararse de forma automática por nosotros, y así ahorrarnos esos seis caracteres?

02. ¿Podría ser self opcional?

Bruce Eckel propuso en 2008 exactamente eso: que self pasara a ser una keyword y pudiera usarse dentro de una clase, pero que no hubiera que añadirla como primer parámetro en cada método que definamos.

Self in the Argument List: Redundant is not Explicit

`http://www.artima.com/weblogs/viewpost.jsp?thread=239003`

02. Si self fuera opcional

```
1  import math
2
3  class Punto(object):
4      def __init__(x, y):
5          self.x = x
6          self.y = y
7
8      def distancia(otro):
9          x_delta = self.x - otro.x
10         y_delta = self.y - otro.y
11         return math.sqrt(x_delta ** 2 + y_delta ** 2)
```

02. Si self fuera opcional — ventajas

Eliminaría redundancias y errores confusos para principiantes como:

```
1 class Punto(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6 p = Punto(1)
```

```
Traceback (most recent call last):
  File "./code/02/23-self-error.py", line 6, in <module>
    p = Punto(1)
TypeError: __init__() takes exactly 3 arguments (2 given)
```

02. self *no* puede ser opcional

El Zen de Python

Explícito es mejor que implícito.

02. La explicación larga: no es tan sencillo

Hay una justificación de tipo teórico: recibir el objeto actual como argumento refuerza la equivalencia entre dos modos de llamar a un método:

```
1  import math
2
3  class Punto(object):
4      def __init__(self, x, y):
5          self.x = x
6          self.y = y
7
8      def distancia(self, otro):
9          x_delta = self.x - otro.x
10         y_delta = self.y - otro.y
11         return math.sqrt(x_delta ** 2 + y_delta ** 2)
12
13 c1 = Punto(4, 1.5)
14 c2 = Punto(3, 3.1)
15 print c1.distancia(c2) == Punto.distancia(c1, c2)
```

```
True
```

02. Un argumento más práctico

La referencia explícita a self hace posible añadir una clase dinámicamente, añadiéndole un método sobre la marcha.

02. Añadiendo un método dinámicamente

```
1  class C(object):
2      pass
3
4  def init_x(clase, valor):
5      clase.x = valor
6
7  C.init_x = init_x
8
9  c = C()
10 c.init_x(23)
11 print c.x
```

```
23
```

init_x() está definida fuera de la clase (es una función), y como argumento recibe la referencia al objeto cuyo atributo x es modificado. Basado en el ejemplo de Guido van Rossum.

02. No puede ser opcional — otra razón

Pero el argumento definitivo está en los decoradores: al decorar una función, no es trivial determinar automáticamente si habría que añadirle el parámetro self o no: el decorador podría estar convirtiendo la función en, en entre otros, un método estático (que no tiene self) o en un método de clase (classmethod, que recibe una referencia a la clase, no al objeto).

Arreglar eso implicaría cambios no triviales en el lenguaje, considerar casos especiales y un poco de magia negra. Y hubiera roto la compatibilidad hacia atrás de Python 3.1.

02. El primer parámetro: self

Why explicit self has to stay (Guido van Rossum)

`http://neopythonic.blogspot.com.es/2008/10/why-explicit-self-has-to-stay.html`

Moraleja

Usamos self para acceder los atributos del objeto [*]

[*] Un método es un atributo ejecutable — ver callable()

THE TWO STATES OF EVERY PROGRAMMER

I AM A GOD.

I HAVE NO IDEA WHAT I'M DOING.

Índice

03. Variables de clase (estáticas)

Una variable estática es aquella ubicada estáticamente (es decir, la memoria se ha reservado en tiempo de compilación) y cuyo tiempo de vida se extiende durante toda la ejecución del programa.

En Python, las variables definidas dentro de una clase, pero no dentro de un método, son llamadas variables estáticas o de clase.

03. Ejemplo

```
1 class T1000(object):
2     jefe = "Skynet"
3
4 robot1 = T1000()
5 robot2 = T1000()
6
7 print robot1.jefe
8 print robot2.jefe
```

```
Skynet
Skynet
```

03. Variables de clase

Estas variables son compartidas por todos los objetos de la clase.

Si ahora asignamos un valor a jefe en objeto de la clase T1000, hemos creado el atributo en ese objeto, pero no hemos cambiado la variable de la clase. Esto ocurre porque el ámbito (*scope*) en el que hemos definido este atributo no es el mismo que el de la clase.

03. Ejemplo — de nuevo

```
1  class T1000(object):
2      jefe = "Skynet"
3
4  robot1 = T1000()
5  robot2 = T1000()
6
7  # Esto no cambia T1000.jefe
8  robot2.jefe = "James Cameron"
9
10 print robot1.jefe
11 print robot2.jefe
```

```
Skynet
James Cameron
```

03. Definiendo constantes

Hay quien acostumbra, viniendo de otros lenguajes de programación, a definir constantes dentro de una clase:

```
1 class Constantes(object):
2     pi = 3.141592653589793
3
4 print "Pi:", Constantes.pi
```

```
Pi: 3.14159265359
```

En Python no necesitamos ninguna clase: tan sólo tenemos que definir nuestras constantes como variables globales (*module-level variables*) — en mayúsculas, de acuerdo con el PEP 8.

03. Definiendo constantes — mejor así

```
1  PI = 3.141592653589793
2
3  class Circulo(object):
4      def __init__(self, x, y, radio):
5          self.x = x
6          self.y = y
7          self.radio = radio
8
9      def area(self):
10         return PI * (self.radio ** 2)
11
12 c = Circulo(0, 0, 4.5)
13 print "Pi:", c.area()
```

```
Pi: 63.6172512352
```

03. Una mala idea

Hay quien podría pensar (sobre todo viniendo de otros lenguajes de programación) que las variables de clase se podrían usar para declarar los atributos de la clase, y asignarles sus valores en __init__().

```
1  class Punto(object):
2
3      x = 0
4      y = 0
5
6      def __init__(self, x, y):
7          self.x = x
8          self.y = y
```

03. Una mala idea — ejemplo

Esto no es buena idea:

- Estamos mezclando variables estáticas con las de objeto.
- No usa más espacio (sólo hay una definición de esos None).
- No es más lento (sólo si la búsqueda de una variable falla).
- Pero puede enmascarar errores.

Is it a good practice to declare instance variables as None in a class in Python?

`https://programmers.stackexchange.com/q/254576`

03. Una mala idea — demostración

```
1  class Circulo(object):
2
3      x = 0
4      y = 0
5      radio = 0
6
7      def __init__(self, x, y, radio):
8          self.x = x
9          self.y = y
10         self.raido = radio # typo!
11
12 c = Circulo(1, 3, 4)
13 print "Radio:", c.radio
```

```
Radio: 0
```

03. Acceso

Para acceder a una variable de clase podemos hacerlo escribiendo el nombre de la clase o a través de self.

La ventaja de self es que hace nuestro código más reusable si en algún momento tenemos que renombrar la clase.

03. Acceso — ejemplo

```
1  # encoding: UTF-8
2
3  class T1000(object):
4
5      JEFE = "Skynet"
6
7      def saluda_uno(self):
8          print self.JEFE, "me envía a eliminarte"
9
10     def saluda_dos(self):
11         print T1000.JEFE, "me envía a eliminarte"
12
13 robot = T1000()
14 robot.saluda_uno()
15 robot.saluda_dos()
```

```
Skynet me envía a eliminarte
Skynet me envía a eliminarte
```

03. Variables de clase (estáticas)

Static class variables in Python

`https://stackoverflow.com/q/68645/184363`

Moraleja

Usamos las variables estáticas (o de clase) para los atributos que son comunes a todos los atributos de la clase. Los atributos de los objetos se definen en `__init__()`

Índice

04. _foo y __spam

Las variables que comienzan por un guión bajo (_foo) son consideradas privadas. Su nombre indica a otros programadores que no son públicas: son un detalle de implementación del que no se puede depender — entre otras cosas porque podrían desaparecer en cualquier momento.

Pero nada nos impide acceder a esas variables.

04. Ejemplo

```
1  class Persona(object):
2
3      def __init__(self, nombre, secreto):
4          self.nombre = nombre
5          self._secreto = secreto
6
7  p = Persona("Raquel", "Prefiere Perl")
8  print "Nombre:", p.nombre
9  print "Secreto:", p._secreto
```

```
Nombre: Raquel
Secreto: Prefiere Perl
```

04. Privadas — por convención

El guión bajo es una señal: nos indica que no deberíamos acceder a esas variables, aunque técnicamente no haya nada que nos lo impida.

"We're all consenting adults here" [*]

[*] `https://mail.python.org/pipermail/tutor/2003-October/025932.html`

04. Es posible, pero no debemos

Nada te impide hacerlo, pero se asume que todos somos lo suficientemente responsables. No se manipulan las variables internas de otras clases u objetos.

Python prefiere obviar esa (en gran parte) falsa seguridad que keywords como protected o private dan en Java — el código fuente siempre se puede editar, al fin y al cabo, o podemos usar reflexión.

04. Algo que no se debe hacer — ejemplo

```
1  class Persona(object):
2
3      def __init__(self, nombre, secreto):
4          self.nombre = nombre
5          self._secreto = secreto
6
7  p = Persona("Raquel", "Prefiere Perl")
8  print "Secreto:", p._secreto
9
10 p._secreto = "Programa en Notepad"
11 print "Secreto:", p._secreto
```

```
Secreto: Prefiere Perl
Secreto: Programa en Notepad
```

04. Evitando colisiones

Las variables privadas también son útiles cuando queremos tener un método con el mismo nombre que uno de los atributos de la clase.

No podemos tener un atributo y un método llamados, por ejemplo, edad: en ese caso es común almacenar el valor en un atributo privado (Persona._edad)

04. Evitando colisiones — ejemplo

```
1  # encoding: UTF-8
2
3  class Persona(object):
4
5      # Máxima edad confesada en público
6      MAXIMA_EDAD = 35
7
8      def __init__(self, nombre, edad):
9          self.nombre = nombre
10         self._edad = edad
11
12     def edad(self):
13         return min(self._edad, self.MAXIMA_EDAD)
14
15 p = Persona("Juan Pedro", 41)
16 print "Nombre:", p.nombre
17 print "Edad:", p.edad()
```

```
Nombre: Juan Pedro
Edad: 35
```

04. Variables ocultas — *enmarañadas*

Dos guiones bajos al comienzo del nombre (__spam) llevan el ocultamiento un paso más allá, enmarañando (*name-mangling*) la variable de forma que sea más difícil verla desde fuera.

04. Variables ocultas — ejemplo

```
1  class Persona(object):
2
3      def __init__(self, nombre, edad):
4          self.nombre = nombre
5          self.__edad = edad
6
7  p = Persona("Juan Pedro", 23)
8  print "Nombre:", p.nombre
9  print "Edad:", p.__edad
```

```
Nombre: Juan Pedro
Edad:
Traceback (most recent call last):
  File "./code/04/43-mangled-example-0.py", line 9, in <module>
    print "Edad:", p.__edad
AttributeError: 'Persona' object has no attribute '__edad'
```

04. Accediendo a __spam

Pero en realidad sigue siendo posible acceder a la variable. Tan sólo se ha hecho un poco más difícil.

Podemos acceder a ella a tráves de _clase__attr, donde clase es el nombre de la clase actual y attr el nombre de la variable. Hay casos legítimos para hacerlo, como, por ejemplo, en un depurador.

04. Accediendo a __spam — ejemplo

```
1  class Persona(object):
2
3      def __init__(self, nombre, edad):
4          self.nombre = nombre
5          self.__edad = edad
6
7  p = Persona("Juan Pedro", 23)
8  print "Edad:", p._Persona__edad
```

```
Edad: 23
```

04: Conflicto de nombres

El objetivo de las __variables es prevenir accidentes, evitando colisiones de atributos con el mismo nombre que podrían ser definidos en una subclase.

Por ejemplo, podríamos definir en una clase heredada el mismo atributo (_PIN) que la clase base, por lo que al asignarle un valor perderíamos el valor definido por esta última.

04: Conflicto de nombres — ejemplo

```
1 class Habitacion(object):
2     _PIN = 9348
3
4 class CamaraAcorazada(Habitacion):
5     def __init__(self, PIN):
6         self._PIN = PIN
7
8 p = CamaraAcorazada(2222)
9 print "PIN:", p._PIN # sobreescribe PIN de Habitacion
```

```
PIN: 2222
```

04: Conflicto de nombres — solución

Usando __PIN, la variable es automáticamente renombrada a _Habitacion__PIN y _CamaraAcorazada__PIN, respectivamente.

Hemos evitado así la colisión aunque usemos el mismo nombre en la clase base y la subclase.

04: Conflicto de nombres — demostración

```
1  class Habitacion(object):
2      __PIN = 9348
3
4  class CamaraAcorazada(Habitacion):
5      def __init__(self, PIN):
6          self.__PIN = PIN
7
8  p = CamaraAcorazada(2222)
9  print "PIN1:", p._Habitacion__PIN
10 print "PIN2:", p._CamaraAcorazada__PIN
```

```
PIN1: 9348
PIN2: 2222
```

04: Conflicto de nombres — mejorado

```
1  class Habitacion(object):
2      __PIN = 9348
3
4  class CamaraAcorazada(Habitacion):
5      def __init__(self, PIN):
6          self.__PIN = PIN
7
8      def PIN1(self):
9          return self._Habitacion__PIN
10
11     def PIN2(self):
12         return self.__PIN
13
14 p = CamaraAcorazada(2222)
15 print "PIN1:", p.PIN1()
16 print "PIN2:"; p.PIN2()
```

```
PIN1: 9348
PIN2:
```

04. _foo y __spam

En la mayor parte de las ocasiones, lo que necesitamos es _foo. Tan sólo en escenarios muy concretos nos hace falta usar __spam. escenarios muy concretos.

Trivia: el *name-mangling* sólo se produce si el nombre de la variable tiene dos guiones bajos al comienzo y como mucho uno al final. __PIN y __PIN_ se *enmarañarían*, pero no __PIN__. Ése es el motivo por el que podemos usar los métodos mágicos (__init__() y demás) sin problema.

04. _foo y __spam

Ned Batchelder propuso en 2006 el término "dunder" (de *double underscore*). Así, __init__() sería "*dunder init dunder*", o simplemente "*dunder init*".

Dunder

`http://nedbatchelder.com/blog/200605/dunder.html`

04. _foo y __spam

Private Variables and Class-local References

https://docs.python.org/2/tutorial/classes.html#private-variables-and-class-local-references

Python name mangling: When in doubt, do what?

https://stackoverflow.com/q/7456807/184363

Moraleja

Usamos _foo para variables privadas. En la mayor parte de las ocasiones, __spam no hace falta. Ante la duda, mejor sólo un guión bajo.

Índice

05. Métodos estáticos

Los método estáticos (*static methods*) son aquellos que no necesitan acceso a ningún atributo de ningún objeto en concreto de la clase.

05. *self* innecesario — ejemplo

```
1  class Calculadora(object):
2
3      def __init__(self, nombre):
4          self.nombre = nombre
5
6      def modelo(self):
7          return self.nombre
8
9      def suma(self, x, y):
10         return x + y
11
12 c = Calculadora("Multivac")
13 print c.modelo()
14 print c.suma(4, 8)
```

```
Multivac
12
```

05. Métodos estáticos

En este caso no es necesario que recibamos como primer argumento una referencia al objeto que está llamando el método.

Podemos usar @staticmethod, un decorador, para que nuestra función no lo reciba.

05. *self* innecesario — ejemplo

```
1  class Calculadora(object):
2
3      def __init__(self, nombre):
4          self.nombre = nombre
5
6      def modelo(self):
7          return self.nombre
8
9      @staticmethod
10     def suma(x, y):
11         return x + y
12
13 c = Calculadora("Multivac")
14 print c.modelo()
15 print c.suma(4, 8)
```

```
Multivac
12
```

05. Métodos estáticos

Nada nos impediría mover este método a una función fuera de la clase, ya que no hace uso de ningún atributo de ningún objeto, pero la dejamos dentro porque su lógica (hacer sumas) pertenece conceptualmente a Calculadora.

Los métodos estáticos no dependen de las características individuales de los objetos que los invocan, por lo que podemos considerar que en realidad pertenecen "a la clase" (siempre que no los confundamos con los classmethods). Eso también significa que los podemos llamar directamente sobre la clase, sin que haya que crear un objeto de la misma.

05. Métodos estáticos — sobre la clase

```
1  class Calculadora(object):
2
3      def __init__(self, nombre):
4          self.nombre = nombre
5
6      def modelo(self):
7          return self.nombre
8
9      @staticmethod
10     def suma(x, y):
11         return x + y
12
13 print Calculadora.suma(4, 8)
```

```
12
```

05. Métodos de clase

Los métodos de clase (*class methods*) pueden visualizarse como una variante de los métodos normales: sí reciben un primer argumento, pero la referencia no es al objeto que llama al método (self), sino a la clase de dicho objeto (cls, por convención).

05. Métodos de clase — motivación

Una referencia a la clase es lo que nos hace falta si necesitamos devolver otro objeto de la misma clase.

Por ejemplo, podríamos tener la clase Ameba y querer implementar su método fisión — que simula el proceso de bipartición y devuelve dos objetos de la misma clase.

05. Ejemplo — *hard-codeando* el tipo

```
1  # encoding: UTF-8
2
3  class Ameba(object):
4
5      def __init__(self, nombre):
6          self.nombre = nombre
7
8      def fision(self):
9          h1 = Ameba("Primogénito")
10         h2 = Ameba("Benjamín")
11         return h1, h2
12
13 ameba = Ameba("Foraminifera")
14 hijo1, hijo2 = ameba.fision()
15 print "Hijo 1:", hijo1.nombre
16 print "Hijo 2:", hijo2.nombre
```

```
Hijo 1: Primogénito
Hijo 2: Benjamín
```

05. Problema: *hard-coding* + herencia

```
1  # encoding: UTF-8
2
3  class Ameba(object):
4
5      def __init__(self, nombre):
6          self.nombre = nombre
7
8      def fision(self):
9          h1 = Ameba("Primogénito")
10         h2 = Ameba("Benjamín")
11         return h1, h2
12
13 class AmebaCyborg(Ameba):
14     pass
15
16 ameba = AmebaCyborg("Foraminifera T-800")
17 hijo1, hijo2 = ameba.fision()
18 print "Hijo 1 es", type(hijo1)
19 print "Hijo 2 es", type(hijo2)
```

```
Hijo 1 es <class '__main__.Ameba'>
Hijo 2 es <class '__main__.Ameba'>
```

05. Ejemplo — usando __class__ o type()

```
1  # encoding: UTF-8
2  class Ameba(object):
3
4      def __init__(self, nombre):
5          self.nombre = nombre
6
7      def fision(self):
8          cls1 = self.__class__
9          h1 = cls1("Primogénito")
10         # O también...
11         cls2 = type(self)
12         h2 = cls2("Benjamín")
13         return h1, h2
14
15 ameba = Ameba("Foraminifera")
16 hijo1, hijo2 = ameba.fision()
17 print "Hijo 1:", hijo1.nombre
18 print "Hijo 2:", hijo2.nombre
```

```
Hijo 1: Primogénito
Hijo 2: Benjamín
```

05. Ejemplo — más problemas

type(self) es, de forma casi indiscutible, preferible a acceder a self.__class__

El único escenario en el que no podemos usarlo es en las clases *old-style*, porque antes de la unificación de clase y tipo type() devolvía siempre <type 'instance' >.

05. Ejemplo — *old-style*

```
1  # encoding: UTF-8
2  class Ameba:
3
4      def __init__(self, nombre):
5          self.nombre = nombre
6
7      def fision(self):
8          clase = type(self)
9          print "Mi tipo es:", clase
10         h1 = clase("Primogénito")
11         h2 = clase("Benjamín")
12         return h1, h2
13
14 ameba = Ameba("Foraminifera")
15 ameba.fision()
```

```
Mi tipo es: <type 'instance'>
Traceback (most recent call last):
  File "./code/05/56-new-object-problem-3.py", line 15, in <module>
    ameba.fision()
  File "./code/05/56-new-object-problem-3.py", line 10, in fision
    h1 = clase("Primogénito")
TypeError: must be classobj, not str
```

05. Métodos de clase

Así que, técnicamente, __class__ es la única solución que funciona tanto en clases *old-style* como *new-style*.

Pero todo eso da igual, porque precisamente @classmethod, otro decorador, existe para recibir como primer argumento el tipo (es decir, la clase) del objeto que ha llamado al método.

05. Métodos de clase — ejemplo

```
1  # encoding: UTF-8
2
3  class Ameba(object):
4
5      def __init__(self, nombre):
6          self.nombre = nombre
7
8      @classmethod
9      def fision(cls):
10         h1 = cls("Primogénito")
11         h2 = cls("Benjamín")
12         return h1, h2
13
14 ameba = Ameba("Foraminifera")
15 hijo1, hijo2 = ameba.fision()
16 print "Hijo 1:", hijo1.nombre
17 print "Hijo 2:", hijo2.nombre
```

```
Hijo 1: Primogénito
Hijo 2: Benjamín
```

05. A veces necesitamos type()

Por supuesto, usar type(self) es inevitable cuando necesitamos crear objetos de la misma clase y acceder a atributos del objeto que llama al método.

05. A veces necesitamos type() — ejemplo

```
1  # encoding: UTF-8
2
3  class Ameba(object):
4
5      def __init__(self, nombre):
6          self.nombre = nombre
7
8      def fision(self):
9          cls = type(self)
10         ascendencia = ", hijo de " + self.nombre
11         h1 = cls("Primogénito" + ascendencia)
12         h2 = cls("Benjamín" + ascendencia)
13         return h1, h2
14
15 ameba = Ameba("Foraminifera")
16 hijo1, hijo2 = ameba.fision()
17 print "Hijo 1:", hijo1.nombre
18 print "Hijo 2:", hijo2.nombre
```

```
Hijo 1: Primogénito, hijo de Foraminifera
Hijo 2: Benjamín, hijo de Foraminifera
```

05. Métodos de clase y estáticos

Python @classmethod and @staticmethod for beginner?

https://stackoverflow.com/a/12179752/184363

Python type() or __class__, == or is

https://stackoverflow.com/a/9611083/184363

05. Métodos de clase y estáticos

Moraleja (I)

Usamos @staticmethod cuando el método trabaja con la clase, no con sus objetos.

Moraleja (II)

Usamos @classmethod cuando trabajamos con la clase y queremos devolver un objeto de la misma clase.

Being a Game Developer

how i thought it would be

how it usually is

Índice

06. *getters* — Por qué

Consideremos la clase Circulo, con los atributos radio y area.

El área depende del radio: hay una relación entre ambos atributos, por lo que si cambiamos uno de ellos habría que recalcular el otro. El área es una función del radio.

06. *getters* — Problema

```
1  # encoding: UTF-8
2
3  import math
4
5  class Circulo(object):
6
7      def __init__(self, radio):
8          self.radio = radio
9          self.area = math.pi * radio ** 2
10
11 c = Circulo(1)
12 print "Radio:", c.radio
13 print "Área:", c.area
14 print
15
16 c.radio = 2
17 print "Radio:", c.radio
18 print "Área:", c.area
```

```
Radio: 1
Área: 3.14159265359
Radio: 2
Área: 3.14159265359
```

06. *getters*

La solución es sencilla: podemos convertir area en una función, area(), y hacer que calcule el área del círculo dependiendo del radio de nuestro objeto en cada momento.

06. *getters* — Ejemplo

```
1  # encoding: UTF-8
2  import math
3
4  class Circulo(object):
5
6      def __init__(self, radio):
7          self.radio = radio
8
9      def area(self):
10         return math.pi * self.radio ** 2
11
12 c = Circulo(1)
13 print "Radio:", c.radio
14 print "Área:", c.area()
15 c.radio = 2
16 print "Radio:", c.radio
17 print "Área:", c.area()
```

```
Radio: 1
Área: 3.14159265359
Radio: 2
Área: 12.5663706144
```

06. *getters*

Funciona, pero hemos pasado de leer un atributo a llamar a un método.

06. *setters* — Por qué

A este círculo le podemos poner radio negativo.

06. *setters* — Problema

```
1  class Circulo(object):
2
3      def __init__(self, radio):
4          self.radio = radio
5
6  c = Circulo(13)
7  print "Radio:", c.radio
8  c.radio = -2
9  print "Radio:", c.radio
```

```
Radio: 13
Radio: -2
```

06. *setters*

Podríamos usar una variable privada para almacenar el radio, y proporcionar métodos en la clase tanto para acceder a la variable como para ajustar su valor.

Es ahí donde haríamos la comprobación de que el valor es no negativo, lanzando un error en caso de que así sea.

06. *setters* — Ejemplo

```
1  # encoding: UTF-8
2  class Circulo(object):
3
4      def __init__(self, radio):
5          self._radio = radio
6
7      def set_radio(self, radio):
8          if radio < 0:
9              raise ValueError("'radio' debe ser un número no negativo")
10         self._radio = radio
11
12 c = Circulo(3.5)
13 print "Radio:", c._radio
14 c.set_radio(-2)
15 print "Radio:", c._radio
```

```
Radio: 3.5
Traceback (most recent call last):
  File "./code/06/603-setter-example-1.py", line 14, in <module>
    c.set_radio(-2)
  File "./code/06/603-setter-example-1.py", line 9, in set_radio
    raise ValueError("'radio' debe ser un número no negativo")
ValueError: 'radio' debe ser un número no negativo
```

06. Mutadores

Estos métodos son conocidos como *setters* y *setters*

También llamados mutadores, son métodos que usamos para controlar los accesos y cambios de una variable. De esta forma, podemos implementar cualquier funcionalidad en el acceso a los atributos de la clase: desde validar los nuevos datos (como en el ejemplo anterior) o disparar un evento cuando una variable sea leída.

06. Mutadores — Ejemplo

```
1  # encoding: UTF-8
2
3  class CajaFuerte(object):
4
5      def __init__(self, PIN):
6          self._PIN = PIN
7
8      def get_PIN(self):
9          print "Enviando copia a la NSA..."
10         return self._PIN
11
12 hucha = CajaFuerte(7821)
13 print "PIN:", hucha.get_PIN()
```

```
PIN: Enviando copia a la NSA...
7821
```

06. Propiedades

En Python, las propiedades nos permiten implementar la funcionalidad exponiendo estos métodos como atributos.

De esta forma, podemos seguir trabajando leyendo el valor de atributos o asignándoles un nuevo valor, pero entre bastidores se están ejecutando funciones que controlan el acceso. Esto mola mil.

06. Propiedades — Ejemplo

```
1  # encoding: UTF-8
2
3  class CajaFuerte(object):
4
5      def __init__(self, PIN):
6          self.PIN = PIN
7
8      @property
9      def PIN(self):
10         print "Enviando copia a la NSA..."
11         return self._PIN
12
13     @PIN.setter
14     def PIN(self, PIN):
15         if len(str(PIN)) != 4:
16             raise ValueError("'PIN' ha de tener cuatro dígitos")
17         self._PIN = PIN
18
19 hucha = CajaFuerte(7814)
20 print "PIN:", hucha.PIN
21 hucha.PIN = 880
```

06. Propiedades — Ejemplo

```
PIN: Enviando copia a la NSA...
7814
Traceback (most recent call last):
  File "./code/06/605-properties-example.py", line 21, in <module>
    hucha.PIN = 880
  File "./code/06/605-properties-example.py", line 16, in PIN
    raise ValueError("'PIN' ha de tener cuatro dígitos")
ValueError: 'PIN' ha de tener cuatro dígitos
```

06. Propiedades

@property convierte PIN() en un *getter* para el atributo de sólo lectura con ese mismo nombre.

Nótese como la asignación en __init__() llama al *setter*, ya que está asignando un valor al atributo PIN — que es ahora una propiedad.

06. area como atributo (*getter*)

```
1  # encoding: UTF-8
2
3  import math
4
5  class Circulo(object):
6
7      def __init__(self, radio):
8          self.radio = radio
9
10     @property
11     def area(self):
12         return math.pi * self.radio ** 2
13
14 c = Circulo(1)
15 print "Radio:", c.radio
16 print "Área:", c.area
17 print
18
19 c.radio = 2
20 print "Radio:", c.radio
21 print "Área:", c.area
```

06. area como atributo (*getter*)

```
Radio: 1
Área: 3.14159265359
Radio: 2
Área: 12.5663706144
```

06. radio como atributo (*setter*)

```
1  # encoding: UTF-8
2
3  class Circulo(object):
4
5      def __init__(self, radio):
6          self.radio = radio
7
8      @property
9      def radio(self):
10         return self._radio
11
12     @radio.setter
13     def radio(self, radio):
14         if radio < 0:
15             raise ValueError("'radio' debe ser un número no negativo")
16         self._radio = radio
17
18 c = Circulo(13)
19 print "Radio:", c.radio
20 c.radio = -2
21 print "Radio:", c.radio
```

06. radio como atributo (*setter*)

```
Radio: 13
Traceback (most recent call last):
  File "./code/06/607-property-fset.py", line 20, in <module>
    c.radio = -2
  File "./code/06/607-property-fset.py", line 15, in radio
    raise ValueError("'radio' debe ser un número no negativo")
ValueError: 'radio' debe ser un número no negativo
```

06. *deleter*

Hay un tercera *property* que podemos crear: el *deleter*

Ahora nuestra clase CajaFuerte guarda un historial de todas las contraseñas, para mayor comodidad de la NSA. El *getter* de PIN nos devuelve la última que se ha ajustado (o None si no hay ninguna), el *setter* la añade al final de la lista interna y el *deleter* vacía esta lista.

06. *deleter* — Ejemplo

```
1  # encoding: UTF-8
2  class CajaFuerte(object):
3      def __init__(self, PIN):
4          self._PINs = []
5          self.PIN = PIN
6
7      @property
8      def PIN(self):
9          try:
10             return self._PINs[-1]
11         except IndexError:
12             return None
13
14     @PIN.setter
15     def PIN(self, PIN):
16         if len(str(PIN)) != 4:
17             raise ValueError("'PIN' ha de tener cuatro dígitos")
18         self._PINs.append(PIN)
19
20     @PIN.deleter
21     def PIN(self):
22         del self._PINs[:]
```

06. *deleter* — Ejemplo

```
1  # No podemos usar 'import' por los guiones
2  modulo = __import__("608-property-deleter")
3
4  hucha = modulo.CajaFuerte(7814)
5  print "PIN:", hucha.PIN
6  hucha.PIN = 8808
7  print "PIN:", hucha.PIN
8  print "Historial:", hucha._PINs
9  del hucha.PIN
10 print "PIN:", hucha.PIN
```

```
PIN: 7814
PIN: 8808
Historial: [7814, 8808]
PIN: None
```

06. @property

Poniéndonos un poco técnicos, lo que está ocurriendo con el decorador @property es que la función se está convirtiendo en en un *getter* para un atributo de sólo lectura de ese mismo nombre.

Este objeto property nos ofrece a su vez los métodos *getter*, *setter* y *deleter*, que podemos usar como decoradores para copiar la propiedad y ajustar su correspondiente función de acceso.

06. *docstring*

El *docstring* de la @property será aquel que especifiquemos para el *setter*.

06. *docstring* — Ejemplo

```
1  # encoding: UTF-8
2
3  import math
4
5  class Circulo(object):
6
7      def __init__(self, radio):
8          self.radio = radio
9
10     @property
11     def area(self):
12         """Devuelve al área del círculo. """
13         return math.pi * self.radio ** 2
14
15 c = Circulo(3.4)
16 print "Radio:", c.radio
17 print "Área:", c.area
18 print "Docstring:", Circulo.area.__doc__
```

```
Radio: 3.4
Área: 36.3168110755
Docstring: Devuelve al área del círculo.
```

06. @property

Al usar decoradores, hay que asegurarse de que las funciones que usamos tienen todas el mismo nombre que la que hemos definido como *setter* [*]

[*] Es decir, aquella que hemos decorado en primer lugar, con @property

06. @property — Ejemplo

```
1  # encoding: UTF-8
2  class Circulo(object):
3
4      def __init__(self, radio):
5          self.radio = radio
6
7      @property
8      def radio(self):
9          return self._radio
10
11     @radio.setter
12     def set_radio(self, radio):
13         if radio < 0:
14             raise ValueError("'radio' debe ser un número no negativo")
15         self._radio = radio
16
17 c = Circulo(13)
18 print "Radio:", c.radio
```

```
Traceback (most recent call last):
  File "./code/06/610-name-unmatch.py", line 17, in <module>
    c = Circulo(13)
  File "./code/06/610-name-unmatch.py", line 5, in __init__
```

06. property()

La sintaxis del decorador es simplemente una interfaz cómoda para la llamada a la función property().

Esta función recibe *setter*, *getter*, *deleter* y *docstring* (opcionales), y nos devuelve el objeto *property* que asignamos a nuestra clase como si fuera un atributo más. Si lo hacemos así, podemos dar a las diferentes funciones el nombre que queramos.

06. property() — Ejemplo

```
1  class CajaFuerte(object):
2      def __init__(self, PIN):
3          self.PIN = PIN
4
5      def get_PIN(self):
6          return self._PIN
7
8      def set_PIN(self, PIN):
9          print "Enviando copia a la NSA..."
10         self._PIN = PIN
11
12     def delete_PIN(self):
13         self.PIN = None
14
15     PIN = property(get_PIN, set_PIN, delete_PIN, "La clave de acceso")
16
17 hucha = CajaFuerte(7814)
18 print "PIN:", hucha.PIN
19 print "Docstring:", CajaFuerte.PIN.__doc__
```

```
Enviando copia a la NSA...
PIN: 7814
Docstring: La clave de acceso
```

06. property() — Argumentos nombrados

No tenemos que usar todos los argumentos a property(). Si sólo pasamos uno será el *getter*; dos, *getter* y *setter*, y así. Para especificar algunos en concreto que no vayan en ese orden tendríamos que usar argumentos nombrados.

Por ejemplo, podríamos querer PIN pueda ser ajustado y borrado, pero —por seguridad— nunca leído. Intentarlo lanzará la excepción AttributeError: unreadable attribute.

06. property() — Ejemplo

```
1  class CajaFuerte(object):
2
3      def __init__(self, PIN):
4          self.PIN = PIN
5
6      def set_PIN(self, PIN):
7          print "Enviando copia a la NSA..."
8          self._PIN = PIN
9
10     def delete_PIN(self):
11         self.PIN = None
12
13     PIN = property(fset=set_PIN, fdel=delete_PIN)
14
15 hucha = CajaFuerte(7814)
16 del hucha.PIN
17 print "PIN:", hucha.PIN
```

```
Enviando copia a la NSA...
Enviando copia a la NSA...
PIN:
Traceback (most recent call last):
  File "./code/06/612-setter-and-deleter.py", line 17, in <module>
```

06. property()

Por supuesto, si intentamos usar alguno que no hemos definido nos encontraremos con un error.

Por ejemplo, si no hemos definido el *deleter*, intentar borrar la *property* con del lanzará el error AttributeError: can't delete attribute.

06. property() — Ejemplo

```
1  # encoding: UTF-8
2
3  import math
4
5  class Circulo(object):
6
7      def __init__(self, radio):
8          self.radio = radio
9
10     @property
11     def area(self):
12         return math.pi * self.radio ** 2
13
14 c = Circulo(1.5)
15 print "Área:", c.area
16 del c.area
```

```
Área: 7.06858347058
Traceback (most recent call last):
  File "./code/06/613-deleter-error.py", line 16, in <module>
    del c.area
AttributeError: can't delete attribute
```

06. Propiedades — siempre

Entre *@properties* vs *getters* y *setters*, la elección está clara: *@properties*, siempre.

La forma Pythónica de acceder a los atributos es hacerlo directamente, ya que todos ellos son públicos. Las *properties* nos permiten que nuestro código sea inicialmente simple y que más tarde incorporemos lógica sin que el resto del código tenga que cambiar — para el resto del mundo, desde fuera, seguirá siendo un atributo.

06. Propiedades — siempre

Por ejemplo, inicialmente podríamos almacenar el radio de la clase Circulo tal y como nos lo dieran, aceptando cualquier valor. Más adelante, podríamos añadir la comprobación de que sea un número no negativo, sin que sea necesario cambiar nada más — lo que no ocurriría al pasar de atributo a *setter* y *getter*.

06. Uniform Access Principle

Formalmente, la idea es adherirnos al Uniform Access Principle: el acceso a todos los atributos lo hacemos a través de una notación uniforme, al margen de que estén implementados mediante simple almacenamiento (un atributo) o una llamada a un método (propiedades).

06. Propiedades (@property)

"public" or "private" attribute in Python ? What is the best way?

`https://stackoverflow.com/q/4555932/184363`

Python @property versus getters and setters

`https://stackoverflow.com/q/6618002/184363`

Moraleja

No usamos *setters* ni *getters*, sino simple acceso a atributos. Si y cuando necesitemos alguna lógica de acceso, podemos reemplazar estos atributos por funciones y usar @properties.

Índice

07. __str__() y __repr__()

La documentación de Python hace referencia a que el método mágico __str__() ha de devolver la representación "informal" del objeto, mientras que __repr__() la "formal".

¿Eso exactamente qué significa?

07. Imprimiendo un objeto Triangulo

```
1  # encoding: UTF-8
2
3  class Triangulo(object):
4
5      def __init__(self, base, altura):
6          self.base = base
7          self.altura = altura
8
9      @property
10     def area(self):
11         return (self.base * self.altura) / 2.0
12
13 t = Triangulo(2, 3)
14 print "Triángulo:", t
15 print "Área:", t.area
```

```
Triángulo: <__main__.Triangulo object at 0x402b966c>
Área: 3.0
```

07. __str__()

La función __str__() debe devolver la cadena de texto que se muestra por pantalla si llamamos a la función str().

Esto es lo que hace Python cuando usamos print.

07. __str__() — Ejemplo

```
1  # encoding: UTF-8
2
3  class Triangulo(object):
4
5      def __init__(self, base, altura):
6          self.base = base
7          self.altura = altura
8
9      @property
10     def area(self):
11         return (self.base * self.altura) / 2.0
12
13     def __str__(self):
14         msg = "Triángulo de base {0} y altura {1}"
15         return msg.format(self.base, self.altura)
16
17 t = Triangulo(2, 3)
18 print str(t)
19 print "Área:", t.area
```

```
Triángulo de base 2 y altura 3
Área: 3.0
```

07. __str__()

No obstante, es mejor hacerlo sin *hard-codear* el nombre de la clase, para que nuestro código sea más reusable.

Podemos acceder al atributo mágico __name__ de la clase actual, type(self), para obtener el nombre de la clase como una cadena de texto.

07. __str__() — Mejor así

```
1  # encoding: UTF-8
2
3  class Triangulo(object):
4      def __init__(self, base, altura):
5          self.base = base
6          self.altura = altura
7
8      @property
9      def area(self):
10         return (self.base * self.altura) / 2.0
11
12     def __str__(self):
13         clase = type(self).__name__
14         msg = "{0} de base {1} y altura {2}"
15         return msg.format(clase, self.base, self.altura)
16
17 t = Triangulo(2, 3)
18 print t
19 print "Área:", t.area
```

```
Triangulo de base 2 y altura 3
Área: 3.0
```

07. __str__()

El objetivo de __str__() es ser legible.

La cadena que devuelve str() no tiene otro fin que el de ser fácil de comprender por humanos: cualquier cosa que aumente la legibilidad, como eliminar decimales inútiles o información poco importante, es aceptable.

07. __repr__()

El objetivo de __repr__() es ser inequívoco.

De __repr__(), por el otro lado, se espera que nos devuelva una cadena de texto con una representación única del objeto. Nos hace falta, por ejemplo, si estamos depurando un error y necesitamos saber, analizando unos logs, qué es exactamente uno de los objetos de la clase.

07. __repr__()

A esta representación única se accede con la función repr() o las comillas hacia atrás (``).

Si __repr__() no está definido, Python, en vez de dar un error, nos genera una representación automática, indicando el nombre de la clase y la dirección en memoria del objeto.

07. __repr__() — No definido

```
1  class Triangulo(object):
2
3      def __init__(self, base, altura):
4          self.base = base
5          self.altura = altura
6
7      def __str__(self):
8          clase = type(self).__name__
9          msg = "{0} de base {1} y altura {2}"
10         return msg.format(clase, self.base, self.altura)
11
12 t = Triangulo(2, 3)
13 print repr(t)
```

```
<__main__.Triangulo object at 0x402b962c>
```

07. __repr__()

Idealmente, la cadena devuelta por __repr__() debería ser aquella que, pasada a eval(), devuelve el mismo objeto.

Al fin y al cabo, si eval() es capaz de reconstruir el objeto a partir de ella, esto garantiza que contiene toda la infomación necesaria.

07. __repr__() y eval()

```
1  class Triangulo(object):
2
3      def __init__(self, base, altura):
4          self.base = base
5          self.altura = altura
6
7      def __str__(self):
8          clase = type(self).__name__
9          msg = "{0} de base {1} y altura {2}"
10         return msg.format(clase, self.base, self.altura)
11
12     def __repr__(self):
13         clase = type(self).__name__
14         msg = "{0}({1}, {2})"
15         return msg.format(clase, self.base, self.altura)
16
17 t = Triangulo(2, 3)
18 print repr(t)
19 print eval(repr(t))
```

```
Triangulo(2, 3)
Triangulo de base 2 y altura 3
```

07. __repr__() sin __str__()

En caso de que nuestra clase defina __repr__() pero no __str__(), la llamada a str() también devuelve __repr__().

07. __repr__() sin __str__() — Ejemplo

```
1  class Triangulo(object):
2
3      def __init__(self, base, altura):
4          self.base = base
5          self.altura = altura
6
7      def __repr__(self):
8          clase = type(self).__name__
9          msg = "{0}({1}, {2})"
10         return msg.format(clase, self.base, self.altura)
11
12 t = Triangulo(2, 3)
13 print str(t)
14 print repr(t)
```

```
Triangulo(2, 3)
Triangulo(2, 3)
```

07. __repr__() sin __str__()

Por eso en el primer ejemplo veíamos el nombre de la clase y su dirección en memoria: la llamada a __str__() falló, por lo que Python nos devolvió __repr__().

El único que de verdad tenemos que implementar es __repr__().

07. __str__() y __repr__() — Ejemplo

```
1 import datetime
2
3 ahora = datetime.datetime.now()
4 print "str() :", str(ahora)
5 print "repr():", repr(ahora)
```

```
str() : 2014-11-09 05:12:13.425097
repr(): datetime.datetime(2014, 11, 9, 5, 12, 13, 425097)
```

07. __str__() y __repr__()

Difference between __str__ and __repr__ in Python

http://stackoverflow.com/q/1436703

Moraleja

__repr__() es para desarrolladores, __str__() para usuarios.

Pair Programming Expectation

Reality

Índice

08. collections.namedtuple()

El módulo collections es genial, y una de sus joyas es namedtuple()

Hay ocasiones en las que tenemos necesitamos clases que no son más que contenedores de atributos. Por ejemplo, podríamos definir la clase Punto para almacenar las coordenadas (x, y, z) de un punto en el espacio tridimensional.

08. Ejemplo — sin namedtuple()

```
1  class Punto(object):
2      """ Punto (x, y, z) en un espacio tridimensional. """
3
4      def __init__(self, x, y, z):
5          self.x = x
6          self.y = y
7          self.z = z
8
9  p = Punto(3, 1, 5)
10 print "x:", p.x
11 print "y:", p.y
12 print "z:", p.z
```

```
x: 3
y: 1
z: 5
```

08. collections.namedtuple()

Aunque no necesitemos hacer nada más con estos atributos, hemos tenido que definir un __init__() lleno de aburridas asignaciones.

Y, al no haber definido __str__() ni __repr__(), ni siquiera podemos imprimir por pantalla nuestro objeto de la forma que esperaríamos.

08. Ejemplo — sin __repr__()

```
1  class Punto(object):
2      """ Punto (x, y, z) en un espacio tridimensional. """
3
4      def __init__(self, x, y, z):
5          self.x = x
6          self.y = y
7          self.z = z
8
9  p = Punto(3, 1, 5)
10 print p
```

```
<__main__.Punto object at 0x402b958c>
```

08. collections.namedtuple()

Tampoco podemos comparar dos puntos sin definir `__eq__()`, ya que sin este método Python comparará posiciones en memoria.

08. Ejemplo — sin __eq__()

```
1  class Punto(object):
2      """ Punto (x, y, z) en un espacio tridimensional. """
3
4      def __init__(self, x, y, z):
5          self.x = x
6          self.y = y
7          self.z = z
8
9      def __repr__(self):
10         args = (type(self).__name__, self.x, self.y, self.z)
11         return "{0}({1}, {2}, {3})".format(*args)
12
13 p1 = Punto(3, 1, 5)
14 p2 = Punto(3, 1, 5)
15
16 print p1, "==", p2, "?"
17 print p1 == p2
```

```
Punto(3, 1, 5) == Punto(3, 1, 5) ?
False
```

08. Ejemplo — con __eq__()

```
1  class Punto(object):
2      """ Punto (x, y, z) en un espacio tridimensional. """
3
4      def __init__(self, x, y, z):
5          self.x = x
6          self.y = y
7          self.z = z
8
9      def __repr__(self):
10         args = (type(self).__name__, self.x, self.y, self.z)
11         return "{0}({1}, {2}, {3})".format(*args)
12
13     def __eq__(self, other):
14         return self.x == other.x and \
15                self.y == other.y and \
16                self.z == other.z
```

08. Esto es muy aburrido

Para algo tan sencillo como imprimir y comparar dos puntos hemos tenido que definir varios métodos.

Es fácil terminar así definiendo incontables métodos mágicos cuando lo único que nosotros queríamos era almacenar tres coordenadas, nada más.

08. collections.namedtuple()

Todo esto podemos ahorrárnoslo usando las *namedtuples*.

08. collections.namedtuple() — Ejemplo

```
1  import collections
2  Punto = collections.namedtuple("Punto", "x y z")
3
4  p = Punto(3, 1, 5)
5  print "x:", p.x
6  print "y:", p.y
7  print "z:", p.z
8
9  p2 = Punto(3, 1, 5)
10 print p2
11 print p == p2
```

```
x: 3
y: 1
z: 5
Punto(x=3, y=1, z=5)
True
```

08. collections.namedtuple()

Lo que namedtuple(), una *factory function*, nos devuelve, es una nueva clase que hereda de tuple pero que también permite acceso a los atributos por nombre.

Nada más y nada menos — pero al ser una subclase significa que todos los métodos mágicos que necesitábamos ya están implementados.

08. collections.namedtuple() — Ejemplo

```
1  # encoding: UTF-8
2
3  import collections
4
5  Punto = collections.namedtuple("Punto", "x y z")
6
7  p = Punto(4, 3, 7)
8  print "x:", p.x
9  print "y:", p[1]
10 print "z:", p[-1]
11 print "Tamaño:", len(p)
```

```
x: 4
y: 3
z: 7
Tamaño: 3
```

08. Creando una *namedtuple*

namedtuple() recibe dos argumentos: el nombre de la clase y sus atributos.

Estos pueden especificarse en una secuencia de cadenas de texto, o en una única cadena de texto en la que estén separados por espacios o comas.

08. Creando una *namedtuple* — Ejemplo

```
1  # encoding: UTF-8
2
3  import collections
4
5  # Todas estas llamadas son equivalentes
6  collections.namedtuple("Punto", ["x", "y", "z"])
7  collections.namedtuple("Punto", "x y z")
8  collections.namedtuple("Punto", "x,y,z")
9  collections.namedtuple("Punto", "x, y, z")
```

08. Nombres de atributos

El nombre de los atributos puede ser cualquier identificador ("*nombre de variable*") válido en Python: letras, números o guiones bajos, pero no pueden empezar por números (esto es normal) ni guión bajo (particulariad de las namedtuples).

Tampoco puede ser ninguna keyword de Python (como for, print o def). De ser así, lanzará ValueError.

08. Nombres de atributos — Ejemplo

```
1 # encoding: UTF-8
2
3 from collections import namedtuple
4 Hucha = namedtuple("Hucha", "altura anchura _PIN")
```

```
Traceback (most recent call last):
  File "./code/08/807-namedtuple-invalid-identifier.py", line 4, in
<module>
    Hucha = namedtuple("Hucha", "altura anchura _PIN")
  File "/usr/lib/python2.7/collections.py", line 341, in namedtuple
    '%r' % name)
ValueError: Field names cannot start with an underscore: '_PIN'
```

08. rename = True

Podemos usar rename = True para que los identificadores inválidos se renombren automáticamente, reemplazándolos con nombres posicionales. También elimina identificadores duplicados, si los hay.

Esto es muy útil si estamos generando nuestra namedtuple a partir de una consulta a una base de datos, por ejemplo.

08. rename = True — Ejemplo

```
1 # encoding: UTF-8
2
3 from collections import namedtuple
4 Hucha = namedtuple("Hucha", "_PIN color for 1", rename=True)
5 print Hucha._fields
```

```
('_0', 'color', '_2', '_3')
```

08. typename

Normalmente asignamos el objeto que namedtuple() nos devuelve (la clase) a un objeto con el mismo nombre.

Esto puede parecer redundante, pero tiene sentido: el primer argumento que le pasamos a namedtuple() es el nombre de la clase, independientemente de qué identificador usemos en nuestro código para referirnos a la misma.

08. typename — Ejemplo

```
1 import collections
2
3 cls = collections.namedtuple("Punto", "x y z")
4 print cls
5 print "Nombre:", cls.__name__
6
7 p = cls(4, 3, 8)
8 print p
```

```
<class '__main__.Punto'>
Nombre: Punto
Punto(x=4, y=3, z=8)
```

08. Añadiendo métodos

Pero con las namedtuples no estamos limitados a la funcionalidad que nos proporcionan. Podemos heredar de ellas para implementar nuestros propios métodos.

08. Añadiendo métodos — Ejemplo

```
1  import collections
2
3  import math
4
5  Punto = collections.namedtuple("Punto", "x y z")
6  class Punto(Punto):
7
8      def distancia(self, other):
9          """ Distancia entre dos puntos. """
10         x_axis = (self.x - other.x) ** 2
11         y_axis = (self.y - other.y) ** 2
12         z_axis = (self.z - other.z) ** 2
13         return math.sqrt(x_axis + y_axis + z_axis)
14
15 p1 = Punto(3, 1, 5)
16 p2 = Punto(5, 2, 7)
17 print "Distancia:", p1.distancia(p2)
18 print
19 print "MRO:", Punto.__mro__
```

```
Distancia: 3.0
MRO: (<class '__main__.Punto'>, <class '__main__.Punto'>, <type
'tuple'>, <type 'object'>)
```

08. Añadiendo métodos

Es preferible heredar de una namedtuple con el mismo nombre pero que comienza por un guión bajo.

Esto es lo mismo que hace CPython cuando un módulo viene acompañado de una extensión en C (por ejemplo, _socket).

08. Añadiendo métodos — Ejemplo

```
1  import collections
2  import math
3
4  _Punto = collections.namedtuple("_Punto", "x y z")
5  class Punto(_Punto):
6
7      def distancia(self, other):
8          """ Distancia entre dos puntos. """
9          x_axis = (self.x - other.x) ** 2
10         y_axis = (self.y - other.y) ** 2
11         z_axis = (self.z - other.z) ** 2
12         return math.sqrt(x_axis + y_axis + z_axis)
13
14 print "MRO:", Punto.__mro__
```

```
MRO: (<class '__main__.Punto'>, <class '__main__._Punto'>, <type
'tuple'>, <type 'object'>)
```

08. Añadiendo métodos

Mejor aún, saltándonos un paso intermedio...

08. Añadiendo métodos — Mejor aún

```
1  import collections
2  import math
3
4  class Punto(collections.namedtuple("_Punto", "x y z")):
5
6      def distancia(self, other):
7          """ Distancia entre dos puntos. """
8          x_axis = (self.x - other.x) ** 2
9          y_axis = (self.y - other.y) ** 2
10         z_axis = (self.z - other.z) ** 2
11         return math.sqrt(x_axis + y_axis + z_axis)
12
13 print "MRO:", Punto.__mro__
```

```
MRO: (<class '__main__.Punto'>, <class '__main__._Punto'>, <type
'tuple'>, <type 'object'>)
```

08. Un ejemplo un poco más completo

```
1  import collections
2  import math
3
4  class Punto(collections.namedtuple("_Punto", "x y z")):
5
6      def distancia(self, other):
7          """ Distancia entre dos puntos. """
8          x_axis = (self.x - other.x) ** 2
9          y_axis = (self.y - other.y) ** 2
10         z_axis = (self.z - other.z) ** 2
11         return math.sqrt(x_axis + y_axis + z_axis)
12
13     def to_zero(self):
14         """ Distancia al origen de coordenadas. """
15         cls = type(self)
16         origen = cls(0, 0, 0)
17         return self.distancia(origen)
18
19 p = Punto(3, 4, 2)
20 print "Distancia:", p.to_zero()
```

```
Distancia: 5.38516480713
```

08. verbose = True

Al llamar a namedtuple(), la opción verbose=True hace que se nos imprima por pantalla la definición de la clase justo antes de construirla.

08. verbose = True — Ejemplo

```
1 import collections
2 Punto = collections.namedtuple("Punto", "x y z", verbose=True)
```

```
class Punto(tuple):
    'Punto(x, y, z)'
    __slots__ = ()
    _fields = ('x', 'y', 'z')
    def __new__(_cls, x, y, z):
        'Create new instance of Punto(x, y, z)'
        return _tuple.__new__(_cls, (x, y, z))
    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Punto object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 3:
            raise TypeError('Expected 3 arguments, got %d' %
len(result))
        return result
    def __repr__(self):
        'Return a nicely formatted representation string'
        return 'Punto(x=%r, y=%r, z=%r)' % self
    def _asdict(self):
        'Return a new OrderedDict which maps field names to their
```

08. _asdict()

El método namedtuple._asdict() nos devuelve un OrderedDict que asociada cada atributo (clave del diccionario) a su valor correspondiente.

08. _asdict() — Ejemplo

```
1 import collections
2
3 Punto = collections.namedtuple("Punto", "x y z")
4 p = Punto(8, 1, 3)
5 print p._asdict()
6
7 for atributo, valor in p._asdict().items():
8     print atributo, "->", valor
```

```
OrderedDict([('x', 8), ('y', 1), ('z', 3)])
x -> 8
y -> 1
z -> 3
```

08. _replace()

Y por último — al heredar de tuple, las namedtuples son, por definición, inmutables. Pero no tenemos por qué abandonar toda esperanza.

Usando el método _replace() podemos cambiar el valor de uno o más atributos, aunque lo que obtenemos es un nuevo objeto (¡ya que el original no puede ser modificado!)

08. _replace() — Ejemplo

```
1  import collections
2
3  Punto = collections.namedtuple("Punto", "x y z")
4  p1 = Punto(8, 1, 3)
5  print "Punto 1:", p1
6
7  p2 = p1._replace(x = 7)
8  print "Punto 2:", p2
9
10 p3 = p1._replace(y = p2.z, z = -1)
11 print "Punto 3:", p3
12
13 print
14 p3.x = 3 # AttributeError
```

```
Punto 1: Punto(x=8, y=1, z=3)
Punto 2: Punto(x=7, y=1, z=3)
Punto 3: Punto(x=8, y=3, z=-1)
Traceback (most recent call last):
  File "./code/08/816-namedtuple-replace.py", line 14, in <module>
    p3.x = 3 # AttributeError
AttributeError: can't set attribute
```

08. collections.namedtuple() — Ventajas

- Proporcionan un punto de partida muy razonable.
- No tenemos que empezar desde cero nuestras clases.
- La inmutabilidad hace nuestro código más seguro.
- Más legibles, más elegantes.
- Menor uso de memoria (__slots__).

08. collections.namedtuple()

What are "named tuples" in Python?

`http://stackoverflow.com/q/2970608/184363`

namedtuple - Python Module of the Week

`http://pymotw.com/2/collections/namedtuple.html`

Moraleja

Usa namedtuple() siempre que puedas.

Índice

09. __init__() y __new__()

Asumamos que vamos a operar con medias ponderadas y queremos almacenar los pesos en su propia clase.

Estos pesos son inmutables, por lo que decidimos heredar de tuple para asegurarnos de que así sea.

09. Una primera aproximación

```
1 class Pesos(tuple):
2     pass
3
4 p = Pesos([0.75, 0.25])
5 print "Pesos:", p
```

```
Pesos: (0.75, 0.25)
```

09. Pero hay un problema

Pronto advertimos un problema: los pesos no tienen por qué sumar uno, como sería de esperar.

Los valores que le pasemos a nuestra clase se van a aceptar sean los que sean.

09. Pero hay un problema — Demostración

```
1 class Pesos(tuple):
2     pass
3
4 p = Pesos([0.50, 0.75, 0.25])
5 print "Pesos:", p
6 print "Total:", sum(p)
```

```
Pesos: (0.5, 0.75, 0.25)
Total: 1.5
```

09. Una posible solución

Así que tenemos una idea: vamos a hacer que `__init__()` compruebe que los valores son correctos y lance `ValueError` de lo contrario.

09. Una posible solución — Ejemplo

```
1  class Pesos(tuple):
2
3      @property
4      def total(self):
5          return sum(self)
6
7      def __init__(self, valores):
8          if sum(self) != 1:
9              msg = "la suma de los pesos ha de ser uno"
10             raise ValueError(msg)
11
12 p1 = Pesos([0.25, 0.75])
13 print "Pesos 1:", p1
14
15 print
16 p2 = Pesos([0.50, 0.75, 0.25]) # ValueError
```

```
Pesos 1: (0.25, 0.75)
Traceback (most recent call last):
  File "./code/09/902-why-we-need-new-2.py", line 16, in <module>
    p2 = Pesos([0.50, 0.75, 0.25]) # ValueError
  File "./code/09/902-why-we-need-new-2.py", line 10, in __init__
    raise ValueError(msg)
```

09. Mejorándolo un poco más

Pero nunca nos damos por satisfechos — ¿y si nuestro constructor, en vez de lanzar un error, se encargara de normalizar los pesos?

Así podríamos darle pesos relativos: [2, 1] sería "*el primer elemento tiene un peso que es el doble que el segundo*", es decir, [0.66, 0.33], de forma cómoda.

09. Mejorándolo un poco más

Podríamos intentarlo así, sin embargo — pero descubrimos que no funciona.

09. Mejorándolo un poco más — Problema

```
1  # encoding: UTF-8
2
3  class Pesos(tuple):
4
5      @property
6      def total(self):
7          return sum(self)
8
9      def __init__(self, valores):
10         valores = [v / self.total for v in valores]
11         super(Pesos, self).__init__(valores)
12
13 p = Pesos([2, 1]) # [0.66, 0.33]... ¿no?
14 print p
```

```
(2, 1)
```

09. ¿Por qué?

Estamos heredando de una clase inmutable, y para cuando llegamos a __init__() ya es demasiado tarde para cambiar nada: el objeto ya ha sido creado.

Y, al ser inmutable, no podemos cambiarlo.

09. __init__() es el *inicializador*

__init__() recibe una instancia (objeto) como su primer argumento (que, no en vano, se llama 'self') y su responsabilidad es inicializarla.

Es decir, ajustar sus atributos a los valores que deban tener. No debe devolver nada.

09. Podríamos intentar hacer esto...

```
1  class Pesos(tuple):
2
3      @property
4      def total(self):
5          return sum(self)
6
7      def __init__(self, valores):
8          valores = [v / self.total for v in valores]
9          self = super(Pesos, self).__init__(valores)
10         return self
11
12 p = Pesos([2, 1]) # sigue sin funcionar
13 print p
```

```
(2, 1)
```

09. ... pero no funciona

Esto no funciona: 'self' es una variable local a la que le hemos asignado un nuevo valor. Y si no ha lanzado error es porque super(Pesos, self).__init__() ha devuelto None y a su vez nuestro __init__().

Ése es el único valor que __init__() puede devolver, porque sólo puede inicializar el objeto — llegados a este punto, el objeto ya existe.

09. Esto tampoco funciona

```
1  # encoding: UTF-8
2
3  class Pesos(tuple):
4
5      @property
6      def total(self):
7          return sum(self)
8
9      def __init__(self, valores):
10         valores = [v / self.total for v in valores]
11         return tuple(valores)
12
13 p = Pesos([2, 1]) # ¡tampoco!
14 print p
```

```
Traceback (most recent call last):
  File "./code/09/905-return-from-init-1.py", line 13, in <module>
    p = Pesos([2, 1]) # ¡tampoco!
TypeError: __init__() should return None, not 'tuple'
```

09. ¿Y qué objeto es ese?

¿De dónde viene el objeto que recibe `__init__()`?
¿Quién lo ha creado?

09. __new__()

El método mágico __new__() recibe la clase como primer argumento, y es su responsabilidad devolver una nueva instancia (objeto) de esta clase.

Es después de llamar a __new__() cuando __init__(), con el objeto ya creado , se ejecuta.

09. Trivia — __init__()

Trivia: el método __init__() de las clases inmutables no hace nada e ignora los argumentos que le pasamos.

Es __new__() quien se encarga de todo. De lo contrario, podríamos usarlo para modificar el valor de objetos inmutables.

09. Trivia — Ejemplo

```
1 # encoding: UTF-8
2
3 variable = "Python"
4 print "Antes  :", variable
5 variable.__init__("Java") # ¡nooo!
6 print "Después:", variable
```

```
Antes  : Python
Después: Python
```

09. Regresando a nuestra clase Pesos

Lo que tenemos que hacer entonces es normalizar los pesos en __new__(), pasándole los valores ya normalizados al método __new__() de la clase base: tuple.

09. Pesos — ahora sí

```
1  from __future__ import division
2
3  class Pesos(tuple):
4
5      def __new__(cls, valores):
6          total = sum(valores)
7          valores = [v / total for v in valores]
8          return super(Pesos, cls).__new__(cls, valores)
9
10     def __init__(self, valores):
11         pass
12
13 print Pesos([2, 1, 3])
```

```
(0.3333333333333333, 0.16666666666666666, 0.5)
```

09. ¡Hay que pasar la clase a __new__()!

```
1  from __future__ import division
2
3  class Pesos(tuple):
4
5      def __new__(cls, valores):
6          total = sum(valores)
7          valores = [v / total for v in valores]
8          return super(Pesos, cls).__new__(valores)
9
10     def __init__(self, valores):
11         pass
12
13 print Pesos([2, 1, 3])
```

```
Traceback (most recent call last):
  File "./code/09/908-Pesos-with-new-1.py", line 13, in <module>
    print Pesos([2, 1, 3])
  File "./code/09/908-Pesos-with-new-1.py", line 8, in __new__
    return super(Pesos, cls).__new__(valores)
TypeError: tuple.__new__(X): X is not a type object (list)
```

09. Aún mejor: con generadores

```
1  from __future__ import division
2
3  class Pesos(tuple):
4
5      def __new__(cls, valores):
6          total = sum(valores)
7          valores = (v / total for v in valores)
8          return super(Pesos, cls).__new__(cls, valores)
9
10     @property
11     def total(self):
12         return sum(self)
13
14 p = Pesos([2, 1, 3])
15 print "Pesos:", p
16 print "Total:", p.total
```

```
Pesos: (0.3333333333333333, 0.16666666666666666, 0.5)
Total: 1.0
```

Y ya no necesitamos __init__() para nada.

09. AnswerToEverything

Otro ejemplo: una clase que siempre devuelve 42, independientemente del valor que se le pase como argumento al crearla.

09. AnswerToEverything — Ejemplo

```
1  # encoding: UTF-8
2
3  class AnswerToEverything(object):
4
5      def __new__(cls, x):
6          print "¡En __new__()!"
7          obj = super(AnswerToEverything, cls).__new__(cls)
8          return obj
9
10     def __init__(self, valor):
11         print "¡En __init__()!"
12         self.valor = 42 # ignora 'valor'
13
14     def __str__(self):
15         return str(self.valor)
16
17 respuesta = AnswerToEverything(23)
18 print "Respuesta:", respuesta
```

```
¡En __new__()!
¡En __init__()!
Respuesta: 42
```

09. AnswerToEverything — Mejor así

```
1  # encoding: UTF-8
2
3  class AnswerToEverything(object):
4
5      def __new__(cls, x):
6          print "¡En __new__()!"
7          return 42
8
9      def __init__(self, valor):
10         # nada que hacer aquí
11         print "¡En __init__()!"
12
13 respuesta = AnswerToEverything(23)
14 print "Respuesta:", respuesta
```

```
¡En __new__()!
Respuesta: 42
```

09. ¿*Constructor* o *inicializador*?

Normalmente nos referimos a __init__() como el *constructor* (incluso en la documentación de Python) pero poniéndonos estrictos quizás deberíamos llamarlo *inicializador*.

09. ¿*Constructor* o *inicializador*?

En realidad el nombre no importa mientras tengamos claro el orden en el que ocurren las cosas:

- Creamos un objeto llamando a la clase.
- __new__() se ejecuta, *args y **kwargs
- __new__() devuelve un nuevo objeto
- __init__() inicializa el nuevo objeto

09. __init__() y __new__()

Python's use of __new__ and __init__?

https://stackoverflow.com/q/674304/184363

Overriding the __new__ method (GvR)

https://www.python.org/download/releases/2.2/descrintro/#__new__

Moraleja

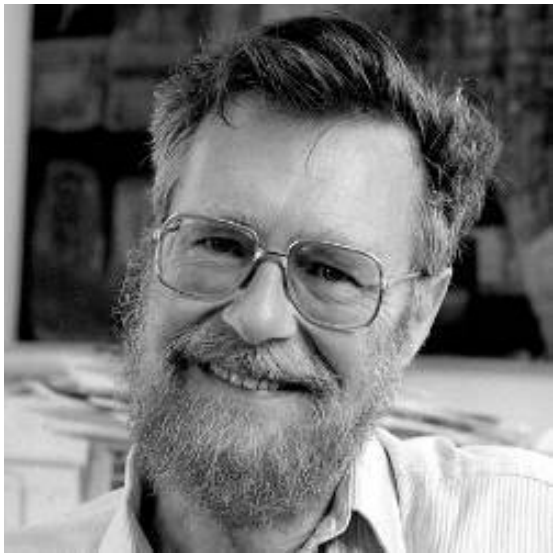
Es __new__() quien crea el nuevo objeto. La mayor parte del tiempo sólo necesitamos tenerlo en cuenta cuando heredamos de clases inmutables.

Índice

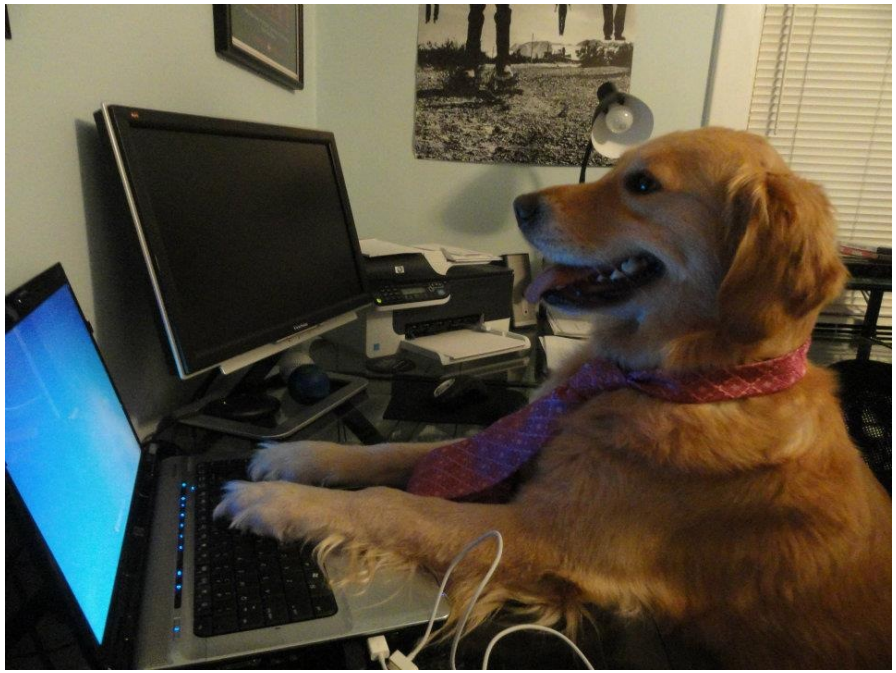
Programación orientada a objetos

Edsger W. Dijkstra

"[...] if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself 'Dijkstra would not have liked this', well, that would be enough immortality for me". [1995, Fuente]



Clases en Python: lo estás haciendo mal



Transparencias y código fuente en:

`http://github.com/vterron/PyConES-2014`